Analysis of Source Code: A Case Study

Danita Hartley
Padmanabhan Krishnan
E-mail: {dah70@student,paddy@cosc}.canterbury.ac.nz

Technical Report TR-COSC 01/01
Department of Computer Science
University of Canterbury, PBag 4800
Christchurch, New Zealand

Abstract

This paper summarises our experience in using model checking technology to understand concurrent programs. We use Verisoft to understand various aspects of a firewall tool kit. We instrument three components of the firewall tool kit with Verisoft hooks in order to test their behaviour. Some of the key changes include changing socket communication to message passing queues and adding appropriate synchronisations so that the behaviour of the system can be tracked. We aim to minimise the number of changes to the original source code so as to not affect its real behaviour. The main conclusion is that it is possible to inspect source code with a view towards verifying key behavioural properties.

1 Motivation

The need for certifying software systems is increasing. However existing standards, such as RTCA DO178-B, are usually limited to highly safety critical software applications. The cost of adopting such a rigorous standard has prevented its uptake in other areas. There is an increased interest in software reliability (especially related to security issues) owing to the internet and web based applications. Customers of not so safety critical systems are also interested in certified software. But the level of certification need not be as stringent as say for avionic systems.

It has been argued that using formal methods will increase the reliability of software [Hei98, Bow94]. A concrete case study is described in [Wid99] where a simple experiment showed the benefits of formal methods. Within formal methods, techniques such as model checking [CGP99] have almost become off the shelf technology. But this is mainly in the context of hardware verification. This is because the requirements for successful use of model checking includes the availability of a finite state model (not exceeding a certain size) expressed in a specification language. Software systems do not usually meet these requirements. They are usually much larger than typical hardware circuits and are written in languages like Java or C. The use of both standard and user defined libraries also imposes a limitation. Thus complete model checking of Java or C programs is not possible.

There are various approaches to reduce the size of the model associated with standard programs. This is usually achieved by some form of static analysis such as slicing, abstraction etc. Tools such as Bandera [CDH$^+$00], ESC/Java based on the work reported in [LN94] perform verification after performing some static analysis. The analysis is usually guided by the property one is interested in verifying. They also rely on the user annotating the program with suitable properties in terms of pre and post conditions.

A different approach is adopted by the tool Verisoft [GHJJ98]. It does not try to compute the complete state space. It analyses the complete run-time behaviour of a program by limiting the depth of the analysis. That is, all reachable states from a given state up to a given depth are examined.

In this paper we describe a case study which uses systematic state space exploration to understand complex programs with a view towards independent product examination. We play the role of an outsider who can examine the source code, test it, change it etc. While we need to change the source code to enable testing, we also aim to minimise such changes. The main aspect we focus on is verification of key behavioural properties. That is, we do not cover all aspects of certification. In particular we do not consider issues such as performance, conformance to standards in a particular operating environment. But what we describe can be extended to address the other issues. What we report is the effort required by a third party to examine software and how this can be used in the certification process.

As the aim is to study a product, we have to examine the source code. One could argue that one should examine a formal specification rather than the source code. Unfortunately, formal specifications are usually not available and secondly, there is no guarantee that the formal specifications correspond to the actual code. Approaches like refinement [Mor94, Abr96], which start with a formal specification and by applying correctness preserving transformations a program is arrived at, have been proposed. However, refinement is not mature enough and not used routinely. Thus, the only option is to examine the code along with other informal supporting documentation.

A brief overview of the paper is as follows. In the next section we describe the firewall

software we are analysing, including the information we have and the type of examination we wish to conduct. We also give a quick overview of the features of our chosen tool, Verisoft. In Section 3 we describe the techniques required to analyse the behaviour of the firewall system described in the previous section. That is we describe the process of instrumenting an analysis with Verisoft, so that we can understand the behaviour of the program. In Section 4 we draw some conclusions and suggest how the process of code inspection can be improved.

2 System and Tools

We chose to analyse the FWTK (FireWall ToolKit) system from Trusted Information Systems (TIS) `http://www.tis.com/research/software`. TIS makes available the sources free of charge for educational and research purposes. Hence it is possible to study the existing code as well as change the code to instrument the analysis. FWTK is organised as a collection of programs including access control (`netacl`), authentication (`auth`), and application programs such as FTP `ftp-gw`, telnet `tn-gw` etc. This organisation makes it easier to inspect the various components independently. These programs are supported by a common library consisting of approximately 2400 lines of C code. In fact, the FWTK documentation states that it is designed to be verified for correctness as a whole or at a component level. FWTK is essentially a concurrent system with processes communicating via sockets. Being a firewall system, it is easy to state some of the key properties the system should satisfy in a simple fashion. For example, safety properties include that a user from a banned host must not be allowed to login. We now describe the key aspects of the access control mechanism `netacl`, the FTP gateway `ftp-gw` and the network authentication service `authd` that are relevant to our analysis.

`Netacl` manages network access control expressed in terms of access form IP-address, service requested etc. `Netacl` is used to block access to services such as FTP and also isolate certain functions using the Unix call `chroot`. It is only about 200 lines of code and, in principle, should be easily analysable. `Netacl` can be configured to allow or deny particular services. For example, the following rule states that one can invoke an FTP server from the host with the specified IP-number.

```
netacl-ftpd: permit-hosts 132.181.11.188 -exec /usr/sbin/in.ftpd -l
```

One of the properties we verify is that a request is accepted only if there is a suitable rule in the configuration file.

`Ftp-gw` is a proxy server for FTP and like `netacl` permits or denies FTP commands based on IP address. It logs all bytes that are transferred for further analysis. The program can be used to run `chroot` to an empty directory and thus cannot be used to break into the firewall. `Ftp-gw` consists of about 1000 lines of code and is larger than `netacl`. Here we do not formally verify any property. Rather we focus on the interaction between the FTP proxy and its operating environment. This will help understand its behaviour.

The authentication service component, `authd`, is an optional component of FWTK and is required if the `ftp-gw` uses authentication. `Authd` essentially maintains a database of users, permissions, failed and successful access etc. on a secured host. We focus on the authentication server `authsrv.c` which is about 1400 lines of code. The man page for the authentication server states what class of users may use particular commands. For example, to use the `group` command the user must be authenticated as a global administrator. We verify a variety of these properties.

Verisoft is a tool that can be used to detect deadlocks and assertion violations in concurrent C programs. While it uses model checking ideas, it does not compute the entire state space of a program. It generates the relevant states up to a given depth from a given state and performs analysis on the generated states. The C programs can communicate via message passing (using message queues established using the Unix system call `msgget`), control access using semaphores (using `semget`, `semctl` calls). One can also write Verisoft specific assertions using the construct `VS_assert(e)` where `e` denotes a boolean expression. If `e` evaluates to false, an error is signalled. One can use `VS_print` to print various messages. Certain operations such as those on message queues, semaphores, assertions and `VS_print` are classified as visible. Hence `VS_print` is different from `printf` in that the former is a hook used by Verisoft. A global state is where all processes can only execute visible operations. The key property that one can analyse deadlocks and assertion violations by only examining the global states is exploited by Verisoft. Verisoft also provides a scheduler which can be used to control the execution of the concurrent program. This allows one to study behaviours under a variety of scheduling algorithms. Any non-determinism that is required can be simulated by the Verisoft operation `VS_toss(n)` which yields a number between 0 and `n`. Verisoft also provides other useful features such as a graphical presentation of the visible operations being performed and the ability to find a scenario leading to an undesirable state.

3 Instrumentation and Analysis

In this section we explain in detail the analysis of FWTK using Verisoft. We begin with a few general remarks on the basic approach. We will present the details later in the section. Our analysis is based on the notion of "as needed comprehension" while performing software inspection [DRW00]. That is, we use Verisoft to verify our understanding of the program. Our understanding of the behaviour of the program is obtained from the man pages and other supporting documents.

As noted earlier, running a program under the control of Verisoft requires the program to be annotated with appropriate visible actions. An important aspect in transforming the code is to make as few changes to the original code as possible, thereby minimising the probability that any new errors are introduced. As communication between processes in Verisoft is limited to message queues and semaphores, all socket communications in FWTK has to be translated using the message passing primitives. While it should be possible to keep the original socket communication and add message passing only for testing purposes, this was not the case for FWTK. The details of this will be explained when we describe the analysis. To minimise changes to the original source code, any function that requires Verisoft additions is called within a *wrapper* function. The interface to a *wrapper* function is kept identical to that of the original function to keep changes minimal. This not only keeps Verisoft code separate from the original source code but also reduces Verisoft code duplication. Ideally the only changes required then are the replacement of particular function names with their Verisoft wrapper equivalents. This can easily be done systematically by a search and replace technique. To further separate the Verisoft additions from the original code, all such additions are kept in a separate file and used in the original source via `#include`.

For socket calls being simulated by Verisoft message passing, there are two cases to consider. In the first case, where the receive from and send to destinations are known, socket simulation is simple. For each socket create two message queues `A` and `B`. If processes 1 and 2

communicate via A and B then process 1 can send data to A and receive data from B. Similarly process 2 can send data to B and receive data from A. If communication is only half duplex only one message queue is required. The second case, where the send to and receive from destinations are not known and communication is full duplex. For example, when socket descriptors are passed as arguments to functions where the functions expect one of several socket descriptors. The problem arises because a single socket identifier becomes two message queue identifiers. In these cases the processes involved know only of one message queue identifier (the queue for receiving data in our case) and the actual queue that is sent to is determined from the receiving queue identifier passed as an argument to the Verisoft wrapper function that does the sending. A socket write call is simulated by a Verisoft `send_to_queue` call while a socket read call is simulated by a Verisoft `rcv_from_queue` call. When data spans several write calls, semaphores are used to force the receiving end to wait until all data has been sent to the queue before any reading from the queue commences.

Function calls that are incompatible with Verisoft need to be replaced by some simulated Verisoft equivalent. For example, the `execv` system call causes a divergence in Verisoft because it does not return to its calling process. We simulate `execv` by executing the body of the program followed by an `exit` system call. This explicitly informs Verisoft that the process in question has now terminated.

We now focus on the processes that need to be created to test the behaviour of the FWTK components. Each component interacts with a client and can also run as a daemon process. To keep the environment as close to reality as possible at least a *client* (also called the *environment*), and a *daemon* process are required. The number of *client* processes required depends on the component. In addition to the environment processes, a process called `process_assert` that keeps track of the state of the main process is implemented. Its purpose is to test if particular properties of the system hold. The process `process_assert` is synchronised with the main process mainly through wrapper functions. For example in the authorisation server, to let the process know when a user is being added to the database the function `auth_dbputu` is wrapped up in a Verisoft wrapper `vs_auth_dbputu` function which simply sends the appropriate message to the assert process and calls the actual `auth_dbputu` function. In certain cases, one has to check the log file to determine the result of an operation. This is because the function being tracked does not return a value. It just makes an entry in the log file. While it is possible to alter the source code to return values, we did not do so in the interests of keeping the changes to a minimum in order to understand the original program.

Simulating an environment process is simple if the interactions with the process being verified (termed as the main process) are understood. The simulation simply consists of sending the appropriate data to the main process and acting appropriately on data received from the main process. When the choice of data sent by an environment process is non-deterministic Verisoft `VS_toss` operations are used. This concludes the description of our approach. We now present details of the verification carried out on the three components of FWTK.

We summarise the general behaviour of `netacl` as specified in the man pages. The man pages form the basis for our verification. `Netacl` runs on different ports for different applications as specified in `/etc/rc.local`. When it receives a request it checks the configuration information and determines if the initiating host has the right permissions. If so, the daemon runs and starts the program specified in the configuration table. It is also possible to `chroot` to a directory and confine the user to a limited area of the system. But the `chroot` command

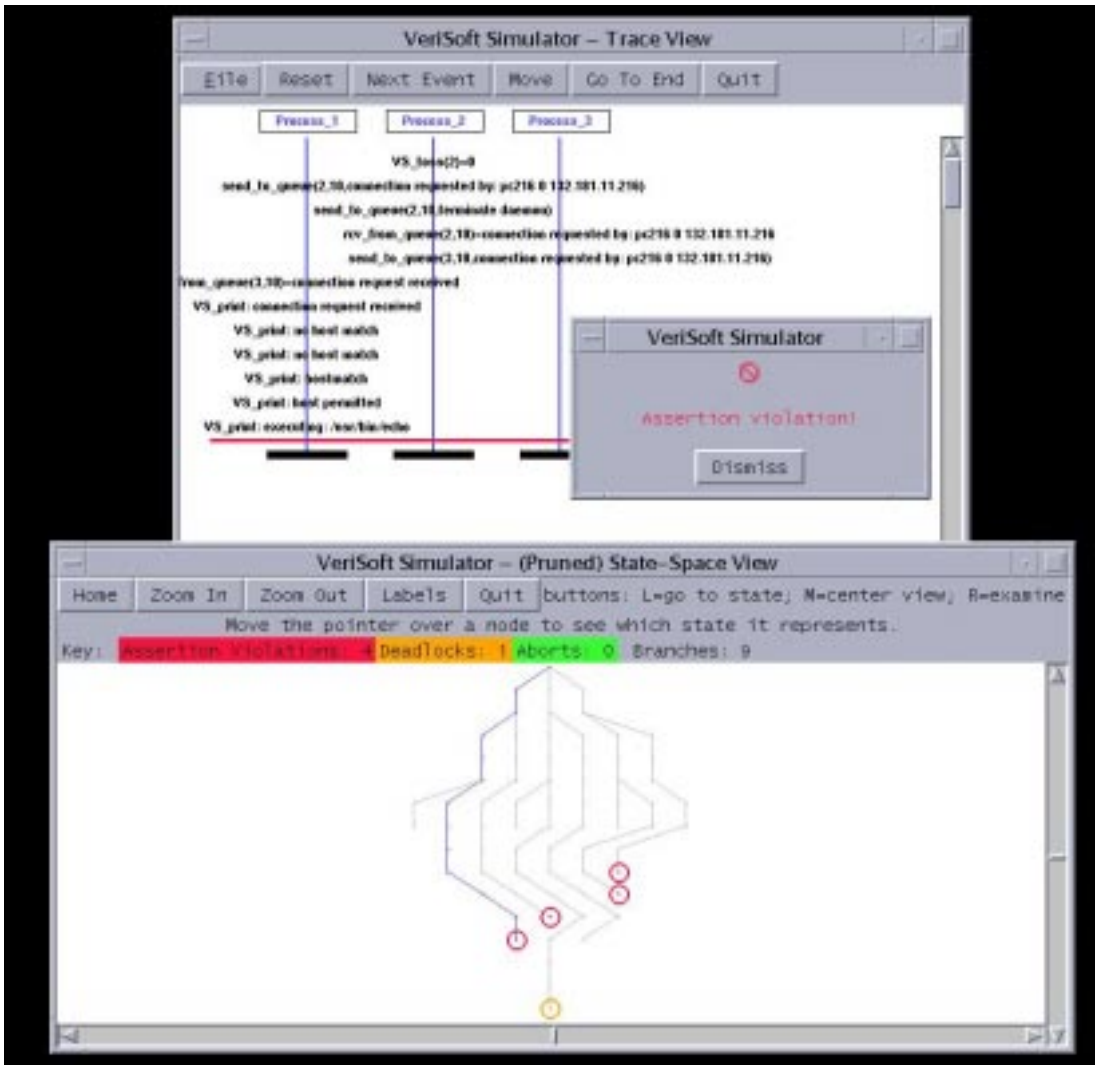

Figure 1: Verisoft detection of failing an assertion in netacl.

can only be executed by those having super-user privileges. We verify the two key properties, viz., a connection is accepted only if a 'permit' rule occurs and a normal user cannot use the 'chroot' option.

In order to verify these properties, two of the standard Unix system calls needed to be modified. The first is `getpeername`, which given a socket, returns the name of the host connected to it. As we are dealing with message queues, there is no equivalent notion. However, we do have a client (the environment) process that generates the request. Hence we can obtain the address from the data generated by the client. This change required a minor change to the `hostmatch` function used by FWTK to verify the permissions. The second is changing `execv` to a Verisoft call so that divergence is not reported. This modified function also asserts that the modified `hostmatch` must have returned true before the required service program was executed. Verisoft had no difficulty in verifying this requirement. If the negation of this assertion was used, Verisoft found a run that violated this assertion.

Consider Figure 1 which shows the details of the result. One deadlock (the lighter circle) and four assertion violations (the darker circles) are shown. The four assertion violations represent the four cases where the environment process is not permitted access. `VS_toss` determines which of the requests one should simulate. The top part of the diagram shows the visible set of actions performed by the processes including the messages that are exchanged which leads to one of the assertion violations. That is, it shows the IP-address that is not permitted in the database and is rejected by the function `hostmatch`. The `VS_print` command is a visible action and is used to indicate which function has been invoked. More specifically, `Process_2` non-deterministically chooses an IP address and sends it on queue 2 to `Process_3`. This request is forwarded on queue 3 to `Process_1`. `Process_1` determines that this host is not permitted to connect and rejects the request. The bottom part of the diagram shows the relevant state space computed by Verisoft. The visible actions that are shown above form a path in the state space graph. In general, Verisoft can be used to determine how a particular state (a point in the pruned state diagram) can be reached. One can click on the point and the sequence of visible actions that lead to that state are displayed.

The deadlock reported is not an error as it indicates the behaviour when a process exists. The sequence of visible actions leading to a deadlock can also be discovered using Verisoft. As shown in Figure 2, Verisoft finds one particular behaviour that leads to deadlock. This is when the a request is denied and the daemon terminates. In this case deadlock is not an error but a required behaviour. The scenario shown is a completion of the behaviour described in Figure 1. After the request has been turned down, the request to terminate the daemon send by `Process_2` is processed by `Process_3`.

For the second property, we tested FWTK in user mode. That is, we focused on the behaviour of a normal user instead of the behaviour of an administrator. A simple assertion after `chroot` was written. Again Verisoft had no difficulty in verifying that one without root privileges could not execute the `chroot` option. Verisoft explored 74 states and 73 transitions. This was based on a depth of 50. Increasing the depth did not affect the number of states as the violation is found within the specified depth.

We now turn our attention to the FTP proxy. The behaviour of the FTP proxy is complicated as it exhibits both security/authentication interactions as well as data transfer related interactions with the server. So we used Verisoft to first understand the various interactions rather than verify any particular property. We study three scenarios to convince ourselves that the FTP proxy works as intended. In the first scenario, the user is not permitted to use the proxy and the program must reject the connection. In the second scenario the user need not be authenticated while in the third scenario the user must be authenticated (using `authsrv`).

Before we describe the scenarios, we outline the changes we had to make to the existing code. In the case of FTP proxy, we needed to modify a number of Unix system calls. These include `select`, `bind`, `listen` etc. These are all related to the socket calls and we have to translate them to equivalent calls on message queues. We also needed to add semaphores to prevent concurrent writing or reading on the same queues especially when a message is split into more than one part. In principle, one can use both message queues and sockets together. That is, sockets would be used as per the original program while message queues would be used only for Verisoft booking keeping. However, this was not possible in this case as it caused a divergence error from Verisoft. This is related to the way the server listens to sockets. As there is no known deadline on when listening succeeds, we needed to change the code to use only message passing.

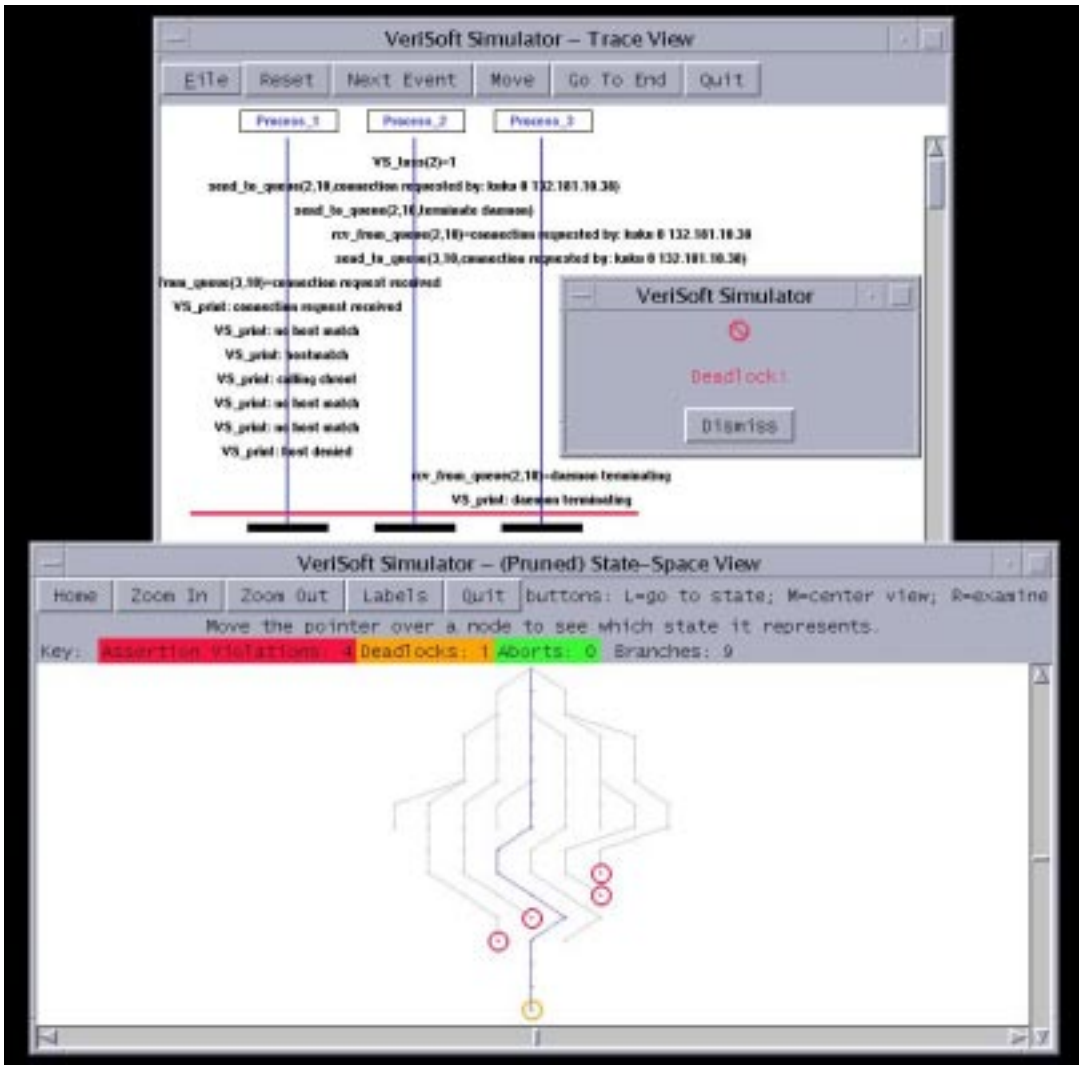

Figure 2: Verisoft finding a deadlock in netacl.

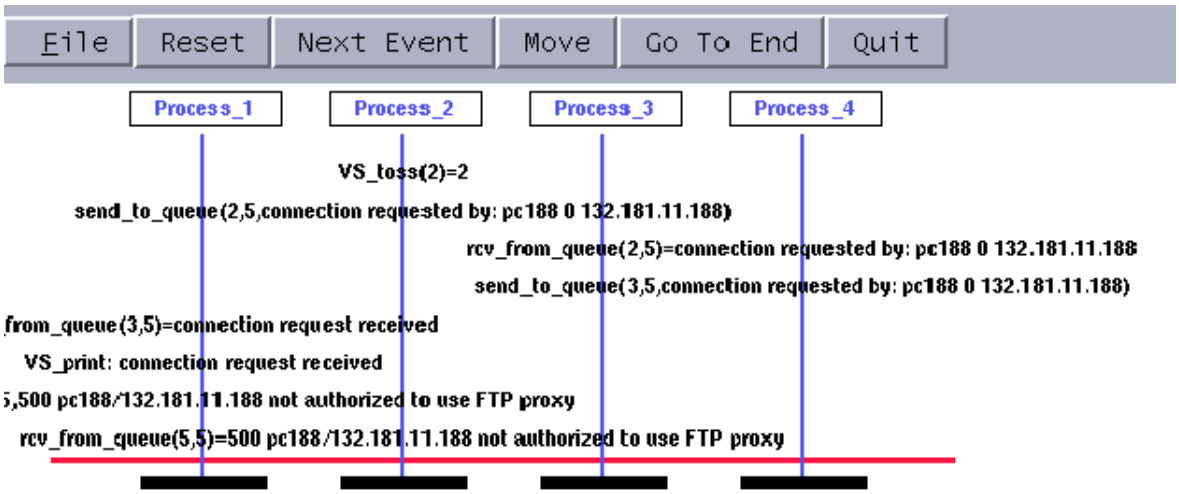

Figure 3: Verisoft scenario of user being denied access.

Consider the behaviour shown in Figure 3. The first process is `ftp-gw`, the second is the environment, the third process is the server while the fourth is the daemon. In this case the server is not activated and thus takes part in no interaction. This behaviour is similar to that seen for `netacl`. As the rules do not allow the host to use the FTP proxy, the request is rejected directly. The interaction shown in Figure 4 shows when a host is permitted. The request is accepted and the proxy is ready to connect to the FTP server. Note that for these behaviours, the server has not taken part in any interaction.

The third scenario is more complicated and is presented in Figure 5. After a request is accepted from a permitted host, the proxy is ready to authenticate the user. The client process simulates a user entering the system and being prompted to send a password. If the password is valid, the user is authenticated to the proxy and the proxy is ready to connect to the server. From a user's perspective this is not surprising. However, it was hard to extract this information from the source code. One of the problems was that the `ftp-gw` uses a table of pointers to functions. Depending on the interaction, a suitable index is computed and the function invoked. It is quite impossible to verify by reading the code that the right functions are invoked. Verisoft helped convince ourselves that the right functions were indeed invoked. Verisoft explored 246 states and 277 transitions before diverging. This is because once the connection is established we do not simulate the behaviour of a user actually interacting with the ftp server.

Our final inspection deals with the authentication server. As noted earlier we need the `process_assert` process. It maintains the state of the authentication server through messages it receives from the server. The process uses its knowledge of the server's state to check if particular properties hold in the current state. To minimise changes to `authsrv` itself, most state messages are sent to `process_assert` via the wrapper functions. For example, whenever a permission denied log message is issued `process_assert` is told permission has been denied and it updates its state and checks various properties hold as appropriate.

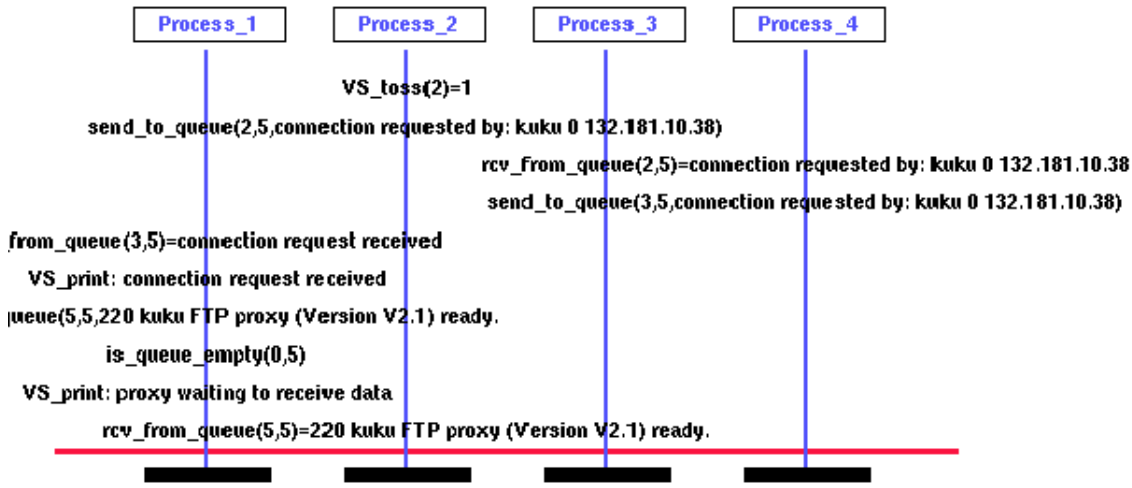

Figure 4: Verisoft scenario of user not requiring authentication.

The properties we tested were influenced by the description of the behaviour in the man pages. For example, it states what class of users may use particular commands. A particular instance is that to use the `group` command the user must be authenticated as a global administrator. All of these permission type properties have been demonstrated to hold (via assertions) on each class of user with one exception. For these analyses, Verisoft explored 408131 states and 515942 transitions. This was to a depth of 100. We also reduced the state space by providing the message passing topology information to Verisoft. We did not do this for the other components, as we were examining the overall behaviour and did not want to pre-judge the communication possibilities.

The man page states the `password` command can only be used by a global administrator or a group administrator of the group to which the user whose password is being changed/set belongs. However, the authentication server also permits any authenticated user to change their own password. This is more likely to be an error in the documentation rather than an error in the code. Figure 6 depicts this error as reported by Verisoft. Here the four processes are as follows. The first two are as before, viz., `authsrv` and the environment respectively. The third process is the daemon while the final process is `process_assert` that is used to synchronise with `authsrv`.

It captures both the interaction that has occurred as well as the location in each process when the assertion error is detected. It is the examination of the source code at this point that led us to believe that the error was in the documentation.

This concludes our discussion of the analysis of three aspects of the FWTK software. In summary, the following were the steps necessary to test various aspects of the firewall.

1. Identify system properties to be checked. These would ideally be documented in the description of the software.

2. If the system has many processes, create a new `main` process to generate all the other processes. This is required by Verisoft.

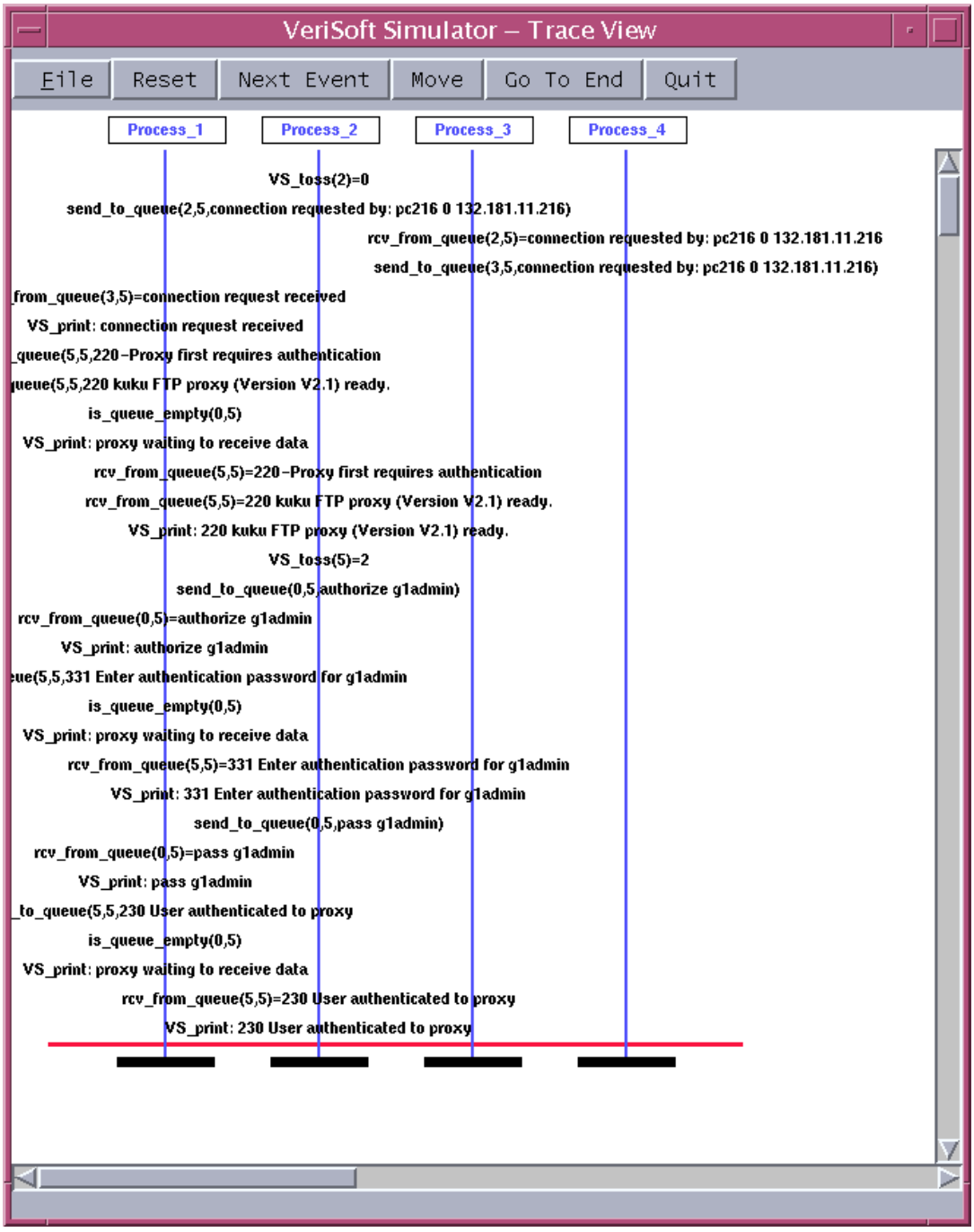

Figure 5: Verisoft scenario of user requiring authentication.

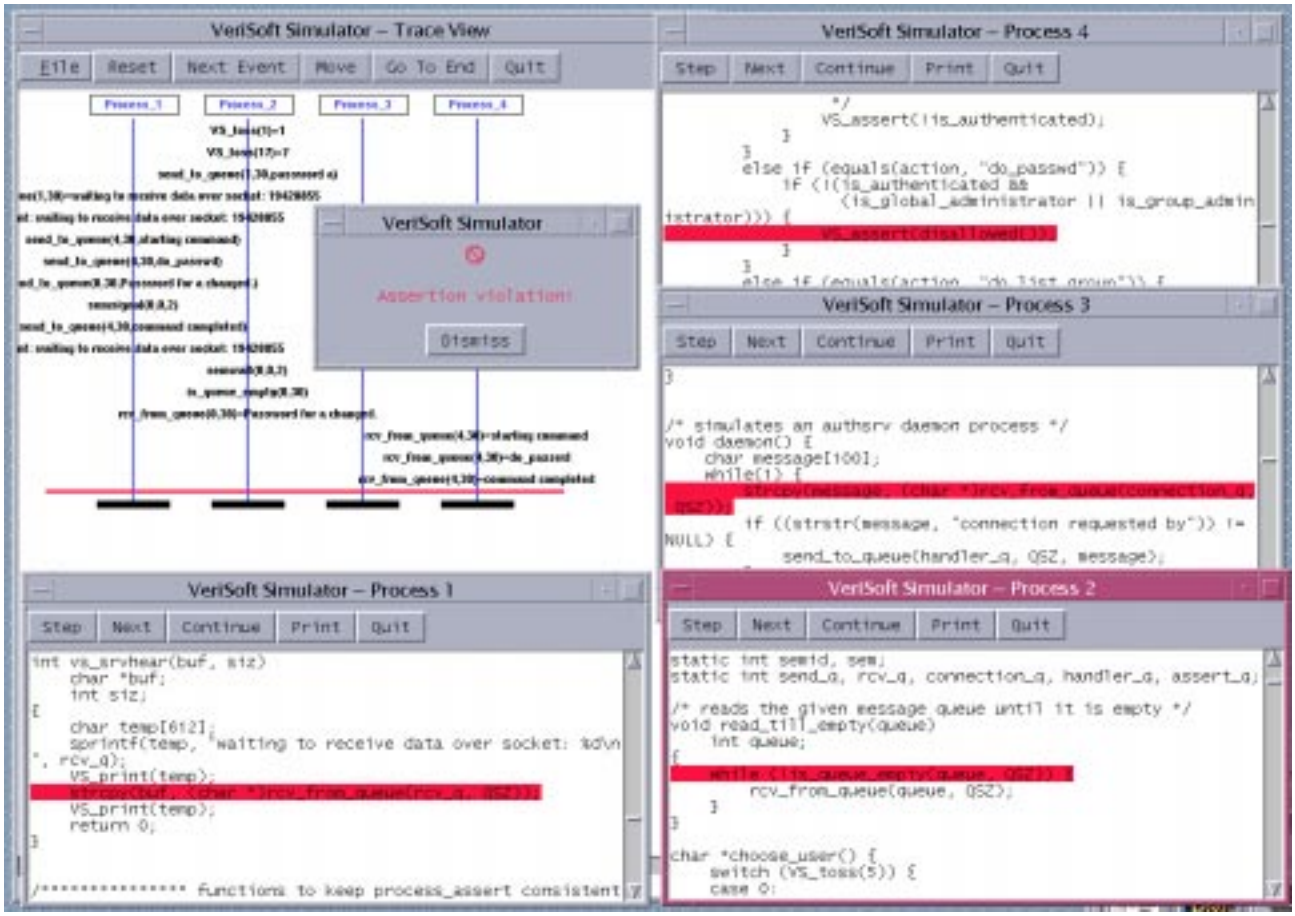

Figure 6: Verisoft detection of the `password` command failing an assertion

3. Identify any type of interprocess communication used and replace with Verisoft message queues. This is likely to involve steps 4 and 5. If the topology of message passing is know, one can provide that information to Verisoft.

4. Identify code incompatible with Verisoft and create *wrapper* functions for this code. For example, system calls such as `read` and `write` which use file descriptors must be instrumented with appropriate wrapper functions that read and write data to message queues. This can be in conjunction with or be replacement for operations involving sockets, file descriptors etc.

5. Replace the original function calls with the wrapper function calls.

6. Implement the processes that model the environment for the processes under analysis. That is, these processes should provide a simulated environment for the main process under analysis.

7. Implement a procedure to track the state of the main process under analysis. This can either be a separate process or some code periodically executed by the main process.

   (a) Introduce additional wrapper functions to report on the state of the main process. These functions will typically send a message to the state tracker process and call the replaced function.

   (b) Add assertions to the state tracker process to verify the identified properties hold.

We now present some of the difficulties faced when trying to use Verisoft. While some of these are due to the requirements of Verisoft, the others pertain to the way the software is written. Verisoft requires that all required message queues must be created before the initial global state is reached. Hence we predefine a large number which are allocated on demand. While this was adequate for our testing, it is not clear if this will always be the case. Similarly, the number of processes should also be fixed and defined in the configuration file. Hence dynamic process creating was not possible. So it was not possible to test multiple rounds of requests from the environment. We could have changed the source code to loop rather than exit (or execute `execv`) to simulate multiple rounds. However, that require more extensive changes to the source code that we were comfortable with. In some cases Verisoft runs into livelock problems. This is due to the way the `vs_select` function, the wrapper for the `select` call needs to implemented. Unfortunately there is no simple solution as there is no bound on when `vs_select` may have anything to select and what it can select. We had to increase it to a large value like 1000 to enable Verisoft to analyse the program. However this was more of a trial and error technique.

Except for concurrency and `VS_toss` operations, Verisoft assumes that the system being tested is deterministic. This assumption produces an error during automatic Verisoft simulation of the authentication server. The actions taken by `authsrv` are dependent on the state of its database file which can be modified during a Verisoft simulation. It is assumed that this violates the above assumption because it is possible that replaying a specific scenario may have a different outcome due to a different database state. For example, the problem arises whenever a user's database account is disabled. The disabling modifies the database state which cannot be reversed during the replaying of scenarios where the user was previously enabled. To make the process deterministic, the database must be re-initialised between paths

in the state space. Since this can not be done during automatic simulation mode, an exhaustive search of the authentication server state space up to some depth when all functionality is examined was not possible. However, an exhaustive search may be possible if the functions that modify the database state are not verified. We could also examine one scenario at a time and reset the database after the analysis of each scenario.

We were also unable to verify conditions such as all transactions are logged. This was because we had no easy access to when a transaction was initiated. The source code does not provide any direct feature that we could use. Note that we could verify well defined aspects of a transaction. For example, conditions such as when a user exits it is logged can be verified. This is because the user exiting is well defined. In general, the code needs to be structured so that Verisoft hooks can be added in a routine way. Extensive changes to the software is both time consuming and has the potential of introducing errors not originally present.

4 Conclusions and Future Work

We were able to gain a thorough understanding of the aspects of FWTK that we studied. We can state with some assurance that the program behaviours we examined are reasonably robust. It has taken about 150 man hours to complete this task. This included installing the Verisoft and FWTK systems, examining the documentation, deciding which aspects to verify and test and actually carrying out the required inspection. Although this is only the first step, the results seems to indicate that independent examination of source code with Verisoft is indeed viable.

One of the main limitations is that we can test only one configuration. The code has various `ifdefs` and we need to automate the testing of all these configurations. We have also not tested all the features of the programs we have examined. For example, there are various ways that authentication can work including DES, one-time passwords, challenge response calculators. We have not examined any of these – we have assumed that the authentication is robust.

More generally, we need to identify coding practices that will simplify the inspection process. These could include, for instance, key functions could be presented as wrapper functions which the inspector can extend, there should be a variable or function that corresponds directly to a key behavioural level concept (such as transactions). We intend to study the Tripwire system (`http://www.tripwiresecurity.com`) to gain further experience and arrive at concrete recommendations.

Acknowledgements

This work has been partially supported by a UoC Summer Scholarship. The authors thank the authors of Verisoft and FWTK for making the tool and the sources available free of charge. Special thanks to Patrice Godefroid for his encouragement and feedback on our work.

References

[Abr96] J. Abrial. *The B Book: Assigning Programs to Meanings.* Cambridge University Press, 1996.

[Bow94] J. Bowen. Formal methods and safety-critical standards. *IEEE Computer*, 27(8):68–71, August 1994.

[CDH+00] J. Corbett, M. Dwyer, J. Hatcliff, C. Pasareanu, Robby, S. Laubach, and H. Zheng. Bandera : Extracting Finite-state Models from Java Source Code. In *Proceedings of the 22nd International Conference on Software Engineering*, June 2000.

[CGP99] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.

[DRW00] A. Dunsmore, M. Roper, and M. Wood. The role of comprehension in software inspection. *The Journal of Systems and Software*, 52(2–3):121–129, June 2000.

[GHJJ98] P. Godefroid, R. Hanmer, and L. Jategaonkar-Jagadeesan. Model checking without a model: An analysis of the heart beat monitor of a telephone switch using verisoft. In *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 124–133, 1998.

[Hei98] C. Heitmeyer. On the Need for *Practical* Formal Methods. In A. P. Ravn and H. Rischel, editors, *Proceedings of the Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume LNCS 1486, pages 18–26, Lyngby, Denmark, 1998. Springer Verlag.

[LN94] K. R. M. Leino and G. Nelson. An Extended Static Checker for Modula-3. In K. Koskimies, editor, *Proceedings of the 1998 Conference on Compiler Construction*, volume 1383 of LNCS, pages 302–305. Springer-Verlag, April 1994.

[Mor94] C. Morgan. *Programming from Specifications*. Prentice Hall, second edition, 1994.

[Wid99] J. Widmaier. Building more reliable software: Traditional software engineering or formal methods? In *Proc. of ISSRE: Industry Practices and Fast Abstracts*, pages 253–260, November 1999.